Control Generation for Embedded Systems Based on Composition of Modal Processes *

Pai Chou, Ken Hines, Kurt Partridge, and Gaetano Borriello

Department of Computer Science and Engineering, Box 352350
University of Washington, Seattle, WA 98195-2350 USA
{chou,hineskj,kepart,gaetano}@cs.washington.edu

Abstract

In traditional distributed embedded system designs, control information is often replicated across several processes and kept coherent by application-specific mechanisms. Consequently, processes cannot be reused in a new system without tailoring the code to deal with the new system's control information. The *modal process* framework [5] provides a high-level way to specify the coherence of replicated control information independently of the behavior of the processes. Thus multiple processes can be composed without internal tailoring and without suffering from errors common in lower-level specification styles. This paper first describes a kernel-language representation for the high-level composition operators; it also presents a synthesis algorithm for the *mode manager*, the runtime code that maintains control information coherence within and between distributed processors.

1 Introduction

To handle the ever-increasing complexity of distributed embedded systems, modern design methodologies must support system *composition*. For this reason, most distributed embedded systems are modeled as communicating *processes*. Process composition has been particularly successful in data-dominated applications, because a set of dataflow processes can be composed as long as they agree on the protocol and data format of their communication.

However, existing process models, based on the idea of *functional decomposition*, do not compose control very well. Control information shared among multiple processes must be encoded as data and communicated using messages. Transmissions, receipts, and tests of control information must then be sprinkled throughout the data-processing code. This approach is error-prone. For example, an update may be accidentally omitted, and deadlock or other synchronization problems may occur. Furthermore, although processes with control information are composable, they are not very modular. Any change involving shared control information requires changing multiple processes [3]. Control-dominated languages such as Esterel [1] and StateCharts [6] attempt to address these problems by also supporting the *temporal decomposition* style of specification. Unfortunately, this results in monolithic, centralized control with no modularity.

Thus, code is rarely reused as is. Since a process must make fixed assumptions about what control interface it wishes to have, it must anticipate the control interactions of any other process with which it is composed. If its interface does not match what is expected by other processes, it cannot be composed with them. Instead, it or some of the other processes must be modified, or an application-specific translation process must be inserted between them. Modification is sometimes impossible for intellectual property reasons, and translation processes tend to be inefficient. Moreover, both techniques require an intimate understanding of what, when, and how control is shared, thus potentially introducing new coherence maintenance errors every time supposedly "reusable" processes are composed.

We introduce the *modal process* framework [5] with an emphasis on enabling *control composition*. Each modal process consists of a set of run-to-completion *handlers* and *modes*. A mode is an enable bit for a set of handlers and is also a basis for spanning the control state space of the system. Rather than keeping modes coherent by communicating their values at the application level, the designer composes the control aspect of the system by applying instances of *abstract control types* (ACTs) to modes of different modal processes. A set of runtime *mode managers* ensures that control is kept coherent on all processes in the system, communicating between themselves as needed. Because the ACTs handle system-level control through mode managers, the modal processes are free to focus on specific modular, reusable behaviors. Modal processes also enhance

*This work was supported by PYI MIP-8858782, DARPA DAAH04-94-G-0272, and a Mentor fellowship.

ICCAD98, San Jose, CA, USA

retargetability by synthesizing the runtime system for a specific distributed target architecture, potentially with different processes-to-processor allocations, without requiring the designer to write low-level synchronization primitives.

This paper describes the semantics of modal processes and the synthesis of mode managers. For synthesis, the coherence requirements are expanded into basic constraint primitives and checked for consistency. Depending on the constraint topology, various optimizations are possible for greater run-time efficiency in terms of both space and time.

2 Programming model

This section describes the two fundamental aspects of our programming model: modal processes and abstract control types. We illuminate this discussion with aspects of a mobile robot example, with control composed from processes for controlling its wheels, its sonar, and its bumper sensor.

2.1 modal processes

A modal process contains a set of code segments called *handlers*, which can be triggered by *events*. Examples of events are notifications of elapsed times and message arrivals. Similar to ROOM [7], the handlers execute with run-to-completion semantics, such that once a handler begins execution, no other handler in that process may execute until it completes. In addition, a modal process also has a number of *modes* that govern the behavior of the process. The state of a mode is called its *status*, which can be either *active* or *inactive*. When a mode is active, it can enable the invocation of a set of handlers to respond to events. A vector that represents the active/inactive status of all modes is known as a *configuration*. Associated with each configuration is a *scheduling policy* that manages the processing of events.

When a handler finishes execution, it may return a request for a configuration change. Changes to the configuration on one modal process may affect the configuration of another modal process. Hence configuration changes are negotiated using a mechanism called a *vote*. In a single-processor architecture, the vote may be processed immediately, but in a distributed architecture, multiple votes may be requested simultaneously, and they must be resolved before being allowed to proceed.

A vote contains a set of pairs, each of which names a mode in the handler's modal process, and the desired new value for the mode. Formally, each component of a vote $v \in V$ is defined to be a member of $M \times \{\text{'+'}, \text{'}-\text{'}\}$, where $+$ means to *activate* the mode (change its status to active), and $-$ means to *deactivate* the mode (change its status to inactive), and M is the set of modes. Any modes in the modal process unmentioned by the vote are treated as "don't cares." However, these modes as well as modes of other modal processes may still be indirectly affected by this vote through composed control.

ACT	input cond.	output cond.
unify($m[1:N]$)	$\pm m[i]$	$\pm m[j]$
mutex($m[1:N]$)	$m[i] \wedge -m[j]$	$-m[j]$
mutexLock($m[1:N]$)	$m[i] \wedge +m[j]$	deny $+m[j]$
parent($p, m[1:N]$)	$\neg p \wedge +m[i]$	$+p$
	$-p$	$-m[1:N]$
(with default)	$+p$	$+m[1]$
guardian($p, m[1:N]$)	$\neg p \wedge +m[i]$	deny $+m[i]$
	$-p$	$-m[1:N]$
(with default)	$+p$	$+m[1]$
preempt($p, m[1:N]$)	$p \wedge +m[i]$	deny $+m[i]$
sequencing($m[1:N]$)	$-m[i]$	$+m[(i\%N)+1]$
seqLoop($s, m[1:N]$)	$+s$	$+m[1]$
	$s \wedge -m[i]$	$+m[(i\%N)+1]$

Table 1: Examples of ACTs

2.2 abstract control types

Control composition is accomplished by means of instantiating abstract control types (ACTs), each of which defines a pattern for constraining how control should flow between a set of modes. This view is similar to the Livingstone [8] approach to reactive self-configuring systems used in the NASA Deep-Space One Probe (DS1) project. However, instead of solving the general satisfiability problem with a fast heuristic for the purpose of reconfiguring the system in response to failed valves, modal processes solve a more restricted problem for the purpose of propagating mode changes imperatively. ACTs can also be prioritized, allowing behavioral composition similar to the subsumption architecture [2]. While handlers are allowed to change only those modes that are local to their process, ACTs allow the local effects to be propagated to other processes globally, as well as customizing the behavior of individual processes.

Some commonly used ACTs are shown in Table 1. The most common way control is composed is to use the `unify` ACT, which correlates modes in different processes and keeps their status the same. In addition, ACTs can relate a set of modes as a flat FSM with the `mutex` ACT, or as superstates/substates with the `parent` ACT; the `sequencing` ACT can be used to express structured control flow. Moreover, `mutexLock` and `guardian` ACTs refine the semantics of `mutex` and `parent` with their ability to *deny* activation requests when locked or when the designated superstate is inactive. The key point is that the framework enables control composition using ACTs as high-level operators that are user definable in terms of simpler ACTs.

example: mobile robot

Consider the example of a bumper process in a mobile robot. The robot normally moves forward until the bumper is hit. Whenever the bumper is hit, the robot should go in reverse until two seconds after the bumper has been released, then it turns 45 degrees before going forward again. Fig. 1(a)

shows a StateChart that captures this behavior. The states are F (forward), B (bumped), W (waiting for 2 second since release), and T (turn).

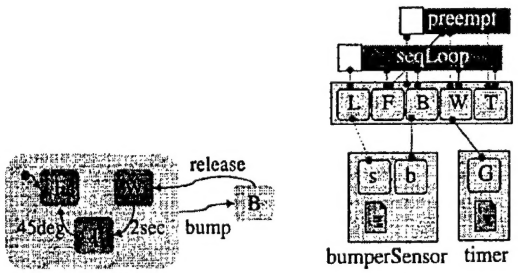

Figure 1: (a) The bumper process described in StateCharts; (b) composition of the bumper process from two reusable components.

The same behavior can be obtained by composition from two reusable components: a bumper sensor and a timer process (Fig. 2) and Table 2. Note that the `preempt` ACT is assigned a higher priority than the `seqLoop`.

mode	handler
bumper sensor	
s	**on** (bumping) vote(+b); **on** (releasing) vote(−b);
b	(no handler)
timer process	
G	**on** (modeEntry) $t := 2$ sec; enableTimer(); **on** (tick) **if** $(--t \leq 0)$ vote(−G); **on** (modeExit) disableTimer();

Figure 2: Reusable modal processes to be composed for the bumper process.

Fig. 3 shows how the composition works. On powerup (Fig. 3(a)), the system initializes to the desired configuration. In this case, mode s should be activated to sense the bumper (bubble (0)). Because L is bound to s and serves as the scope mode in `seqLoop`, +L implies +F, the first body mode of `seqLoop` (bubble (1)), resulting in the configuration shown in Fig. 3(b). When the bumper is hit, mode b is activated by the bumper-sensing handler (bubble (2)). Since b is bound to B and serves as the preempting condition for `preempt`, it forces the deactivation of [F, W, T] (bubble 3). When the bumper is released, the bumper-sensing handler deactivates b (bubble 4), which also deactivates B, and `seqLoop` activates the next mode in sequence, namely W (bubble 5). Since W is unified with the timer's G, it effectively starts the count-down timer. When the timer finishes counting down, it deactivates G (bubble 6) and therefore votes for −W, causing `seqLoop` to activate T for turning (bubble 7). When

composed mode	hidden mode	component process	mode binding
F		(none)	(none)
B		bumperSensor	b
	L		s
W		timerProcess	G
T		(none)	(none)
ACTs for control composition			
`preempt`(B, [F, W, T]) `seqLoop`(L, [F, B, W, T])			

Table 2: Control composition of the bumper process from two reusable components.

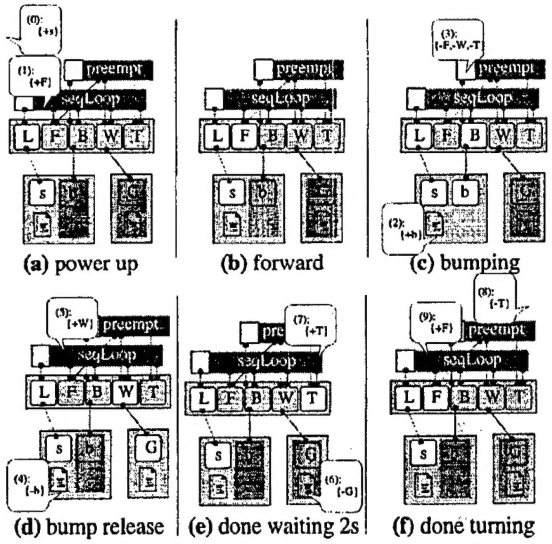

Figure 3: Illustration of operation of the composed bumper process.

turning is completed, the turning handler (not shown) deactivates T mode (bubble 8), causing `seqLoop` to activate F mode (bubble 9). The resulting configuration in Fig. 3(f) is identical to Fig. 3(b).

3 A kernel language

In this section we discuss one perspective on the evaluation semantics for *stateless* ACTs, or ACTs whose behavior is purely functional. The current tool uses a simple *kernel language* for representing ACTs, consisting of only a single simple constraint (Γ) which operates on a sensitivity list, an activity, and an environment. The environment contains consistent configurations for modes in the system. Each primitive constraint is associated with a priority, which may or may not coincide with the evaluation order.

This representation is useful for a couple of reasons: first,

it simplifies the evaluation semantics, and pushes the complexity to the compiler instead of the runtime environment, and second, it simplifies many of the consistency checks that we may want to run on a system before committing to a runtime system.

$$\Gamma : senselist \rightarrow action \rightarrow Z \rightarrow 2^M \rightarrow M \rightarrow env \rightarrow env$$

In the following:

- P represents a modelist of all source modes.
- S represents a senselist representing (parameterized) sensitivity.
- a represents the target mode of the constraint
- A represents the action to be performed on the target mode.
- E represents the environment in which this constraint is evaluated.
- and as indicated above, the final return value is an environment.

$$[[(P,S)\Gamma(a,A)]](E) = if(E \cap S(a) = S, E \uplus (a,A), E)$$

$$A \uplus B = A \cup B - \{\text{values that contradict B}\}$$

Figure 4: Semantics of the primitive constraint.

As shown in Fig 4, the primitive constraint Γ is a function that takes the following curried parameters: a sensitivity list ($\subset senselist = \{(m,p) || m \in$ Modes, $p \in \{'+','-','T','F'\}$), an action to be performed on a mode ($\in action = \{'+','-'\}$), a priority ($\in Z$), an input list ($\in 2^M$), an output ($\in M$) and an environment ($\in env$) and returns an environment.

So–if all of the conditions of the sensitivity list are met by the environment when the constraint is evaluated, then the constraint performs the appropriate action on the environment–but only if this action has not already been preempted by a constraint with a higher priority. Notice that this implies that the conditions in a sensitivity list for a single constraint are related through conjunction. We can achieve a disjunctive relationship by using multiple constraints.

It is important to note that the evaluation order and the priority of constraints are specified separately. A significant effect of this is that constraints can cause changes in the environment that may not appear directly in the new configuration. For example, a particular mode may be associated with activation at some point during the evaluation, and this apparent activation may be propagated through the system – but later this same mode may be deactivated by a constraint with higher priority. This allows modes to be used as temporary place-holders in determining a new configuration.

The curried parameters in the functional definition of the primitive constraint allow us to specialize this for certain general applications. For example, one interesting set of primitive constraints are called the *force* constraints. These

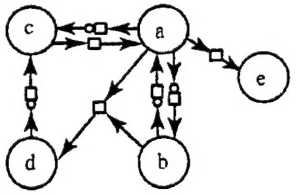

Figure 5: A bipartite digraph representation of modes with primitive constraints.

are constraints with single input and output modes, and are sensitive only to changes. This type of constraint is commonly given a two letter designation indicating the input sensitivity and output action (e.g. AA for activation ('+') sensitivity and an activation action, DA for deactivation ('−') sensitivity and an activation action etc.) Force constraints are actually sufficient for representing many ACTs, so they will appear often in the following discussion.

3.1 graph formulation

A system of modes and primitive constraints can be represented as a bipartite graph $G(M, C, E)$, where M is a set of vertices representing modes, C is a set of vertices representing primitive constraints, and E represents the edges. An example of a graphic representation is shown in figure 5, in which modes are represented by round vertices and constraints are represented by square vertices. Some shorthand is employed in this example: edges entering constraints with small circles indicate '−' sensitivity, edges exiting constraints with small circles indicate a '−' action, and edges without the small circles indicate '+' sensitivity or actions. Note: this graph does not show any information pertaining to evaluation order or to priority. This information was left off for clarity. Also note that most constraints shown in this graph are simple force constraints, with the only exception being the conjunctive constraint between a, b and d (as explained earlier, the evaluation semantics consider the conjunction of all edges entering a constraint vertex, and the disjunction of all edges entering a mode vertex)

3.2 ACT expansion

Building a constraint graph from a set of modes and stateless ACTs is performed by treating each ACT as a constraint macro, and expanding it into its relevant constraints. Both priority and evaluation order are derived from the original ACT description.

As an example, consider the composition of the bumper and wheels processes in Fig. 6. The bumper process is internally constrained as a composition of a `seqLoop`(F, R, W, T) at priority 1 and `preempt`(R, F, W, T) at priority 2. It is possible to apply ACTs across the processes, such as the `or` that designates R and W of the bumper process as the chil-

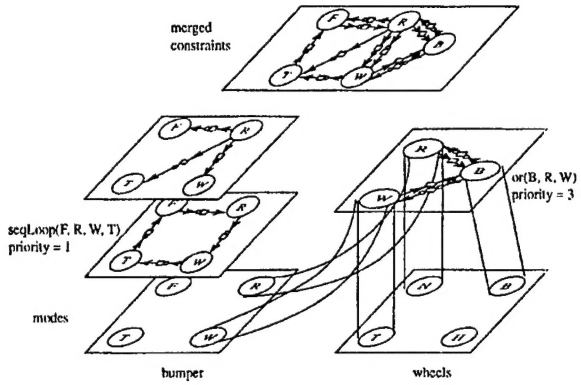

Figure 6: example of ACT expansion on the bumper and wheels processes

dren of mode B of the wheels process. These constraints are merged as shown at the top of Fig 6.

checking consistency

In many applications, the reduction of ACTs to simple constraints makes it possible to perform a variety of consistency checks. An example of one such check is the finding of constraint conflicts, which may result in race conditions. Constraint conflicts occur when the change of one mode propagates through the constraint graph over two separate paths which result in conflicting votes for a single mode. Although this may be desired behavior in some circumstances, such as when the conflict is used to maintain some temporary state, if both paths have the same priority and execution order is arbitrary, this situation may cause an indeterminate system.

To perform a conservative constraint conflict check, it is only necessary to take a transitive closure of the constraints, and compare constraints with a match between left hand side and right hand side arguments.

optimization

There are several optimizations available for a constraint graph, and although many of these depend on the target architecture of the system and will be addressed in later sections, there are some optimizations that may be performed directly when transforming ACTs to constraint graphs. For example, an optimization that may be performed with unification ACTs is to collapse the unified modes into a single *supermode*.

4 Centralized mode manager

Following the transformation of ACTs and constraints into their runtime form, the mode manager code that implements the constraints must be produced. The implementation depends significantly on whether the target architecture is a uniprocessor or a distributed architecture. The discussion of distributed architecture implementations is postponed to the next section.

The centralized mode manager has a notion of a discrete *step*, which defines a sequential boundary for a set of votes to be accumulated and resolved as a single externally visible change of configuration. We support two possible step semantics: event-triggered and time-triggered. Both share the same engine that computes the next configuration.

4.1 computing a new configuration

Algorithm 4.1 Centralized mode manager configuration selection.

foreach vote $V = \{(m_i, s_i, p_i) \in M \times \{+, -\} \times Z\}$
  foreach $((m_i, s_i, p_i) \in V)$
    m_i.set(s_i, p_i)
foreach constraint $C = (\{(m_j, s_j) \in M \times \{+, -, 'T', 'F'\}\}, (m_i, s_i, p_i))$
  ifall m_j.polarity $== s_j$ **and** $p_i > m_i$.priority
    m_i.set(s_i, p_i)

Algorithm 4.1 shows how the next configuration is computed. At the end of a step, the mode manager is given an ordered set of requests, or votes, to change either part or all of the configuration. Each vote is a set of tuples $V = \{(m_i, s_i, p_i) \in M \times \{A, D\} \times Z\}$, where m_i is the specific mode to change, and s_i indicates whether it should be activated or deactivated. Each mode with a pending vote is set to the vote's value, and flagged with the priority of the vote. If there are multiple votes for a single mode, then the mode is set to the value of the highest priority vote. (Since potential votes must be totally ordered, there is always a uniquely chosen vote.)

In the next step, the mode manager evaluates each primitive constraint according to the evaluation order specified by the designer (or the source ACTs) depending on whether or not the constraint has higher priority than the vote already placed on the target mode. Higher priority constraints are always evaluated, even if their result would not conflict with the state of the target since these may change the priority of the state.

Example

To illustrate the operation of the mode manager algorithm, we consider an example based on the bumper process of the robot (Fig. 7). Note that the mode manager maintains the configurations without using any knowledge about what processes the modes belong to. Therefore, the mechanism for managing modes within a process is exactly the same as that for a set of processes.

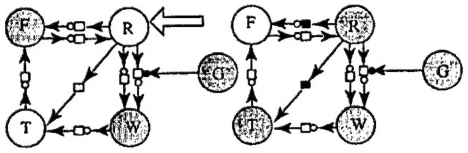

Figure 7: Example for computing the next configuration.

Assume the current configuration is { F, W, G }, and a handler votes for activation of R. The algorithm first marks R active, then iterates over all constraints in the system that are sensitive to this change. These constraints are AD(R, F), AA(R, T), and the conjunctive constraint sensitive to +R and $'F'$G. DA(R, W) is not applicable because the vote is for activation, not deactivation. The result: F is deactivated and T is activated, however, since G is active, the conjunctive constraint is not satisfied and therefore W does not change. The resulting configuration is therefore { R, T, W, G }.

4.2 voting steps

The execution of a modal process system with centralized control can be viewed as a sequence of discrete steps. All events generated during a step are consumed during a later, though not necessarily the next, step. Furthermore, no handler execution crosses a step boundary. Several handlers may be invoked in a given step. If they requests mode changes, the requests are queued until the end of the step when they are processed collectively for the next step. We provide the mechanism for defining a variety of steps, ranging from event-driven steps to dataflow and time-triggered steps.

The simplest step is defined by an event occurrence. That is, the designer may assume no simultaneous events and that a mode change request is serviced right after dispatching an event to a set of handlers. Discrete event models are more general in that events are not only completely ordered but can also be simultaneous, such that vote processing is performed after all (logically) simultaneous events have triggered their handlers.

Another way of defining steps is to mark certain event types as step-delimiting events. For synchronous dataflow (SDF) models, a reasonable step would be to process votes after an entire iteration of the dataflow graph has been invoked. This allows the dataflow graph to be invoked according to a static schedule without using the more expensive event dispatch mechanism. Although dataflow models are untimed, dispatching according to a static schedule can be extended for real-time systems by replacing dataflow events with timer events. In general, statically scheduled, time-triggered systems offer the best determinism and can make the strongest guarantee in meeting hard real-time constraints.

5 Distributed mode manager

When mapping a design to a distributed architecture, control may be implemented in a centralized or a distributed style. If the designer desires a centralized control process, the centralized mode manager described in the previous section can be used, with slight extension to communicate votes explicitly in a message. However, such an organization is not very efficient and defeats the very advantages offered by distributed architectures, because the centralized mode manager must handle and generate communication to all processes even if most are not affected by a localized mode change.

To exploit the architectural distribution, we support *distributed mode managers*, which maintain consistent mode configurations between processes residing on different processors—without centralized control. With distributed mode managers, each processor in the system is given its own mode manager and each of these coordinate activation and deactivations between themselves. In this case, however, there is no single notion of step. In fact, the rate of step progression may be different for each specific mode manager. To avoid over-specification, the modal process model does not impose specific *synchrony* semantics on the interactions between mode managers; instead, several synchrony options can be supported, as described in [4]. This section focuses on one on synchrony option called *mode synchronous* semantics, where a mode change blocks progress of only those processes whose modes are affected until their mode managers agree to it.

The synthesis steps for a distributed mode manager can be divided into graph partitioning, control communication synthesis, and local mode-manager synthesis. Local mode-managers are centralized mode managers whose inputs are their respective partitioned graphs. This section reviews the graph partitioning algorithm that has been described previously and addresses the extensions to the mode managers needed for distributed control coordination.

5.1 mode manager partitioning

In distributed implementations of a modal process system, it isn't necessary for all parts of the system to maintain the complete constraint graph. In fact, each subsystem needs only a projection of the constraint graph containing the portions relevant to the processes in that subsystem. Specifically, these are the modes that occur within these processes (the local modes), and the modes that appear as the source end of a primitive constraint that terminates at a local mode (see Figure 8 *b)* and *c)*). For more information, see [5].

5.2 control communication

Upon completion of the partitioning step, the mode manager residing on each subsystem is aware of which *voted* activations and deactivations of modes need to be transmitted to the rest of the system. Using mode synchronous semantics, the

requesting subsystem is not able to perform the changes until each of the relevant subsystems acknowledges this request. Assuming reliable communication between all mode managers, there is a three phase handshake (request, acknowledge, commit) such that the requester transmits the desired activations and deactivations as a special vote to all relevant mode managers and it waits for the remote subsystems to acknowledge the requests.

The receiving subsystem mode managers include this vote in calculating the next configuration (based on this subsystem's own version of steps). In considering this sort of vote local mode managers must determine whether there were any internally generated conflicting votes—and if there were and they had higher priority than the remote vote, it must send a request of its own to the original requester before acknowledging the original request.

If it finds no such conflict, it simply acknowledges the request and places itself into a provisional state until it receives the corresponding *commit* message. From the perspective of the requester, the actual transition to a new configuration is blocked until all requests have been acknowledged. When the requester receives acknowledgments from all subsystems over all parts of the vote, it performs the action locally (provided there were no conflicts received before getting all acknowledgments) and sends *commit* messages to all relevant subsystems. If it received a conflict in the mean time, and it decided that the conflict had higher priority, then it sends *abort* messages to each of the participants.

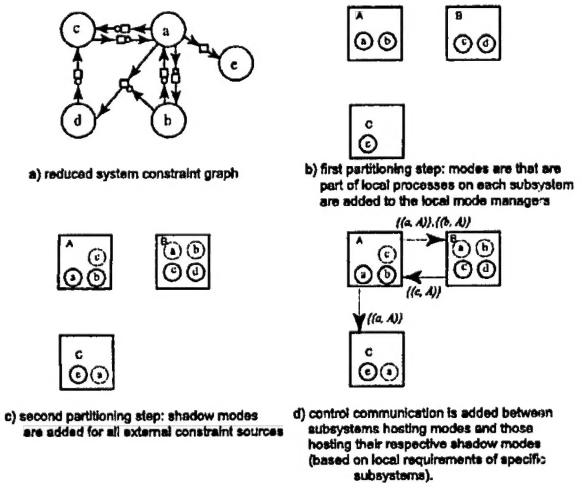

Figure 8: Shown are the steps involved in partitioning the constraint graph for individual mode managers (based on a preexisting process partition), and in synthesizing control communication.

To insure consistent choices in the event of several concurrent requests, there should be system wide total ordering of priorities that allow all subsystems to independently but

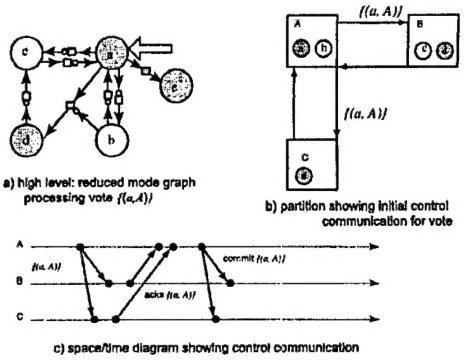

Figure 9: a) shows a system constraint graph, b) shows the mapping of nodes from this to processes in a distributed implementation and c) shows the control communication required to insure mode synchrony.

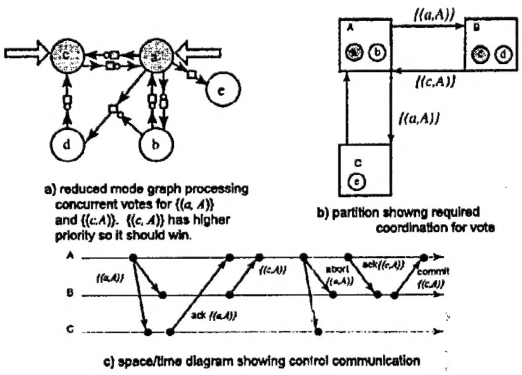

Figure 10: This shows the control communication between the processes in Figure 9 in the presence of an inter-process vote conflict.

consistently choose from among conflicting requests.

Identification of the required control communication is straightforward given the partitioning step. Any constraint such that the source mode is on one subsystem, and the terminal mode is on another implies a communication.

All subsystem mode managers are essentially centralized mode managers, and can be synthesized as demonstrated in the previous section, with some minor modifications.

5.3 Examples

In Figure 8 we show a system constraint graph, and the steps required to build consistent distributed mode managers for this. First the mode graph is partitioned across the subsystems and then the control communication is synthesized.

Next we subject this system to various conditions that might occur in choosing a new consistent configuration (shown in Figures 9 and 10). In Figure 9 the system hosts

a single request for mode activation, and this is easily resolved. In Figure 10 case, there are two concurrent requests for activation, where one request has a higher priority than the the other. In this case, each of the requesting managers must evaluate the relative priority of the requests, and independently (but consistently) choose the winner. The subsystem that requested the losing activation (subsystem A in this case) is then responsible for sending abort messages to all subsystems that received the original request.

6 Results

In this section we present some results from a slightly different implementation of the wall-following robot from the one described in this paper. We look at two implementations - one with a centralized mode manager and one with a distributed mode manager. For the distributed mode manager we look at the communication bandwidth consumed by control communication, and for both we look at the percentage of computation required for maintaining mode consistent modes.

The complete example required forty-five primitive constraints and on average, sixty-three percent of these were ignored during the evaluation step because their priority was less than that of their target mode. For a centralized implementation, on average thirty percent of each scheduling cycle was spent in the mode manager, with occasional peaks of up to fifty percent. It is important to note that this time is not entirely overhead, since the mode manager replaces some activity that would normally be included in the scheduled code.

For a distributed implementation with mode synchronous communication, the average cycle time required by the mode managers rose to approximately fifty percent, with occasional spikes of up to ninety percent.

7 Conclusions

This paper describes a control generation technique for embedded systems specified as a composition of modal processes. Modal processes improve upon traditional programming models in their ability to support control composition using high-level, user-definable operators called abstract control types. The advantages include better re-use of intellectual property and also greatly enhanced retargetability of behavioral specification to heterogeneous distributed architectures. To support this design methodology, implementation techniques are developed for the automatic synthesis of the composed control, called the mode manager, for both single processor and distributed architectures, including all low-level synchronization details. While many implementations of the modal processes abstraction are possible, this paper presented an approach based on a small set of well-defined primitives, or a kernel-language, that can be composed to build up the high-level, user-definable abstractions.

Future work must progress in several directions. The approach here is applicable to *stateless* ACTs, which cover a large class of practical ACTs and enable the generation of highly efficient runtime control, but a more powerful kernel language is needed to represent those ACTs with internal states, such as a mutex that queues requests for serial activation. The mode manager implements composed control by *interpretation* of mode constraints. While this is adequate for most distributed systems where communication cost dominates the overhead, better code generation may be needed to enable low-cost embedded systems to take full advantage of this methodology. An improved user interface will greatly enhance the usability of this methodology. Graphic primitives corresponding to common ACTs can be used to provide an environment where components can be composed and the hierarchy and priority of ACTs can be more intuitively described. Finally, this approach presents new opportunities for formal verification, which may be able to take advantage of the high-level knowledge explicitly specified within the ACTs instead of rederiving it from embedded code. With careful design of the kernel language, it may be possible to extend the idea of composition from control to formal verification.

References

[1] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.

[2] R. A. Brooks and J. H. Connell. Asynchronous distributed control system for a mobile robot. In *Proceedings of the SPIE - The International Society for Optical Engineering*, volume 727, pages 77–84, 1987.

[3] P. Chou. *Control Composition and Synthesis of Distributed Real-Time Embedded Systems*. PhD thesis, University of Washington, 1998.

[4] P. Chou and G. Borriello. An analysis-based approach to composition of distributed embedded systems. In *Proc. International Workshop on Hardware/Software Codesign (CODES/CACHE)*, 1998.

[5] P. Chou and G. Borriello. Modal processes: Towards enhanced retargetability through control composition of distributed embedded systems. In *Proc. Design Automation Conference*, pages 88–93, June 1998.

[6] D. Harel. StateCharts: a visual formalism for complex systems. *Science of Programming*, 8(3):231–274, June 1987.

[7] B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modeling*. Wiley, 1994.

[8] B. C. Williams and P. P. Nayak. A model-based approach to reactive self-configuring systems. In *Proceedings of AAAI-96*, 1996.